EE309 Advanced Programming Techniques for EE

Lecture 16: Synchronization (Advanced)

INSU YUN (윤인수)

School of Electrical Engineering, KAIST

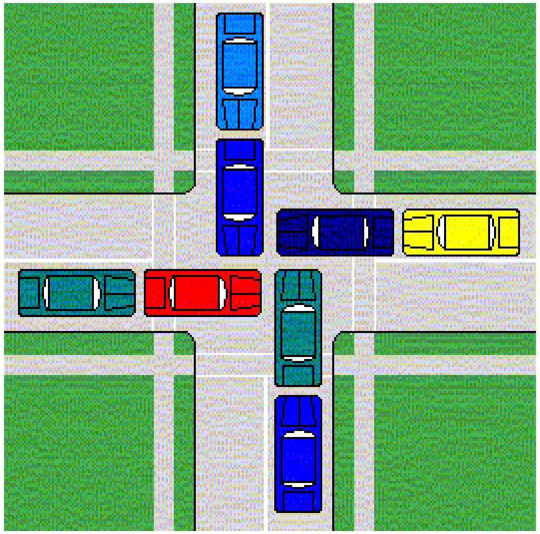
[Slides from 15-213: Introduction to Computer Systems at CMU]

Today

- Deadlock
- Semaphores, Events, and Queues
- Reader-Writer Locks and Starvation
- Thread-Safe API Design

Deadlock

- **A program is *deadlocked* when it is waiting for an event which *cannot* ever happen**
  - Mathematical impossibility, not just practical



- **Most common form:**
  - Thread A is waiting for thread B to do something
  - Thread B is waiting for thread A to do something
  - Neither can make forward progress

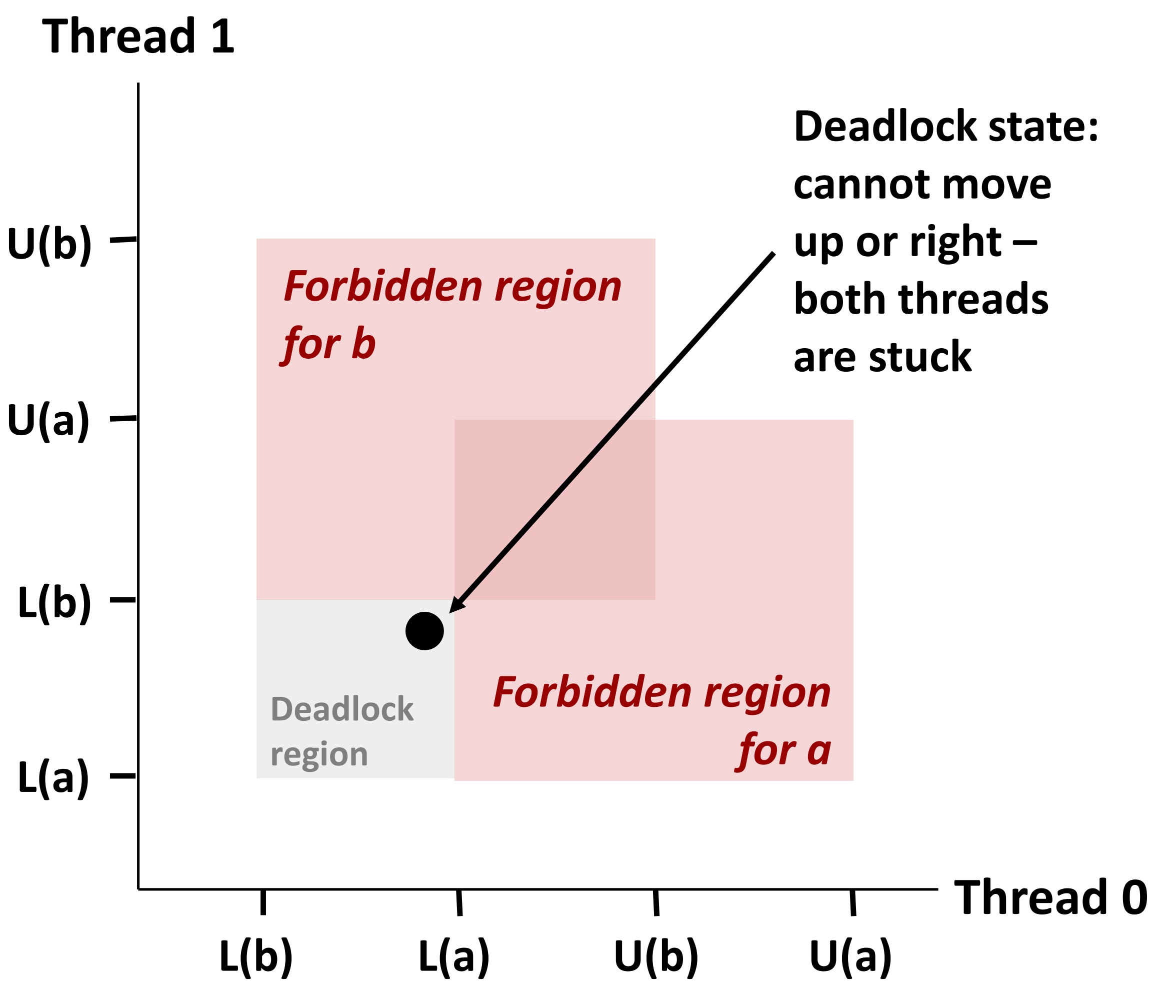
Deadlock caused by wrong locking order

```
void *thread_1(void *arg) {
  pthread_mutex_lock(&mA);
  pthread_mutex_lock(&mB);

  // do stuff ...

  pthread_mutex_unlock(&mA);
  pthread_mutex_unlock(&mB);
}
```

```
void *thread_2(void *arg) {
  pthread_mutex_lock(&mB);
  pthread_mutex_lock(&mA);

  // do stuff ...

  pthread_mutex_unlock(&mB);
  pthread_mutex_unlock(&mA);
}
```

Live coding demo: deadlocks

Deadlock Visualized in Progress Graph



Any trajectory that enters the ***deadlock region*** will eventually reach the ***deadlock state*** where each thread is waiting for the other to release a lock

Other trajectories luck out and skirt the deadlock region

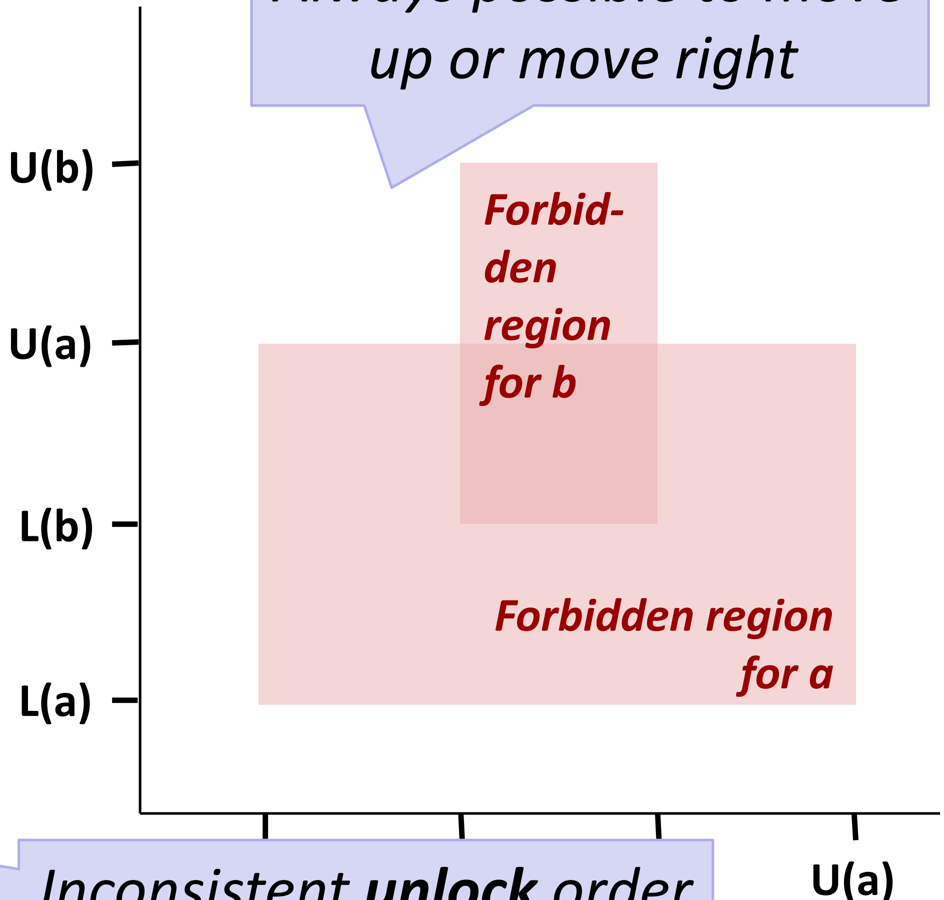
Unfortunate fact: trajectory variations may mean deadlock bugs are nondeterministic (don't always manifest, making them hard to debug)

Fix *this* deadlock with consistent ordering

```
void *thread_1(void *arg) {
  pthread_mutex_lock(&mA);
  pthread_mutex_lock(&mB);

  // do stuff ...

  pthread_mutex_unlock(&mA);
  pthread_mutex_unlock(&mB);
}
```

```
void *thread_2(void *arg) {
  pthread_mutex_lock(&mA);
  pthread_mutex_lock(&mB);

  // do stuff ...

  pthread_mutex_unlock(&mB);
  pthread_mutex_unlock(&mA);
}
```

Always possible to move up or move right



*Inconsistent **unlock** order does not matter*

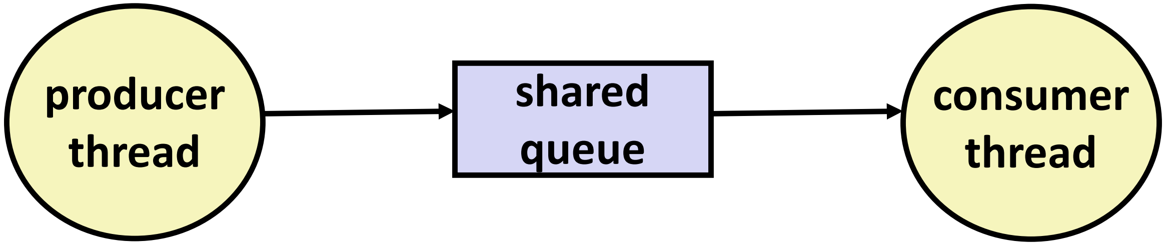
Today

- Deadlock
- **Semaphores, Events, and Queues**
- Reader-Writer Locks and Starvation
- Thread-Safe API Design

Recall: Semaphores

- **Integer value, always >= 0**
- **P(s) operation (aka `sem_wait`)**
  - If *s* is zero, wait for a *V* operation to happen.
  - Then subtract 1 from *s* and return.
- **V(s) operation (aka `sem_post`)**
  - Add 1 to *s*.
  - If there are any threads waiting inside a *P* operation, resume one of them
- **Any thread may call P; any thread may call V; no ordering requirements**
  - Key difference from mutexes

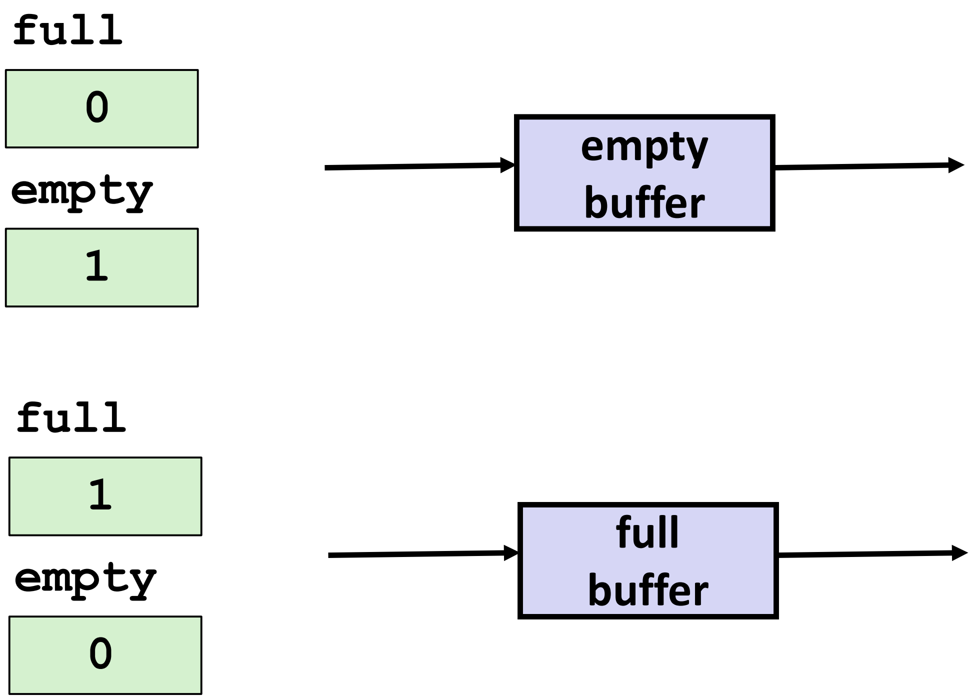
Queues, Producers, and Consumers



- **Common synchronization pattern:**
  - Producer waits for empty ***slot***, inserts item in queue, and notifies consumer
  - Consumer waits for ***item***, removes it from queue, and notifies producer
- **Examples**
  - Multimedia processing:
    - Producer creates video frames, consumer renders them
  - Event-driven graphical user interfaces
    - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in queue
    - Consumer retrieves events from queue and paints the display

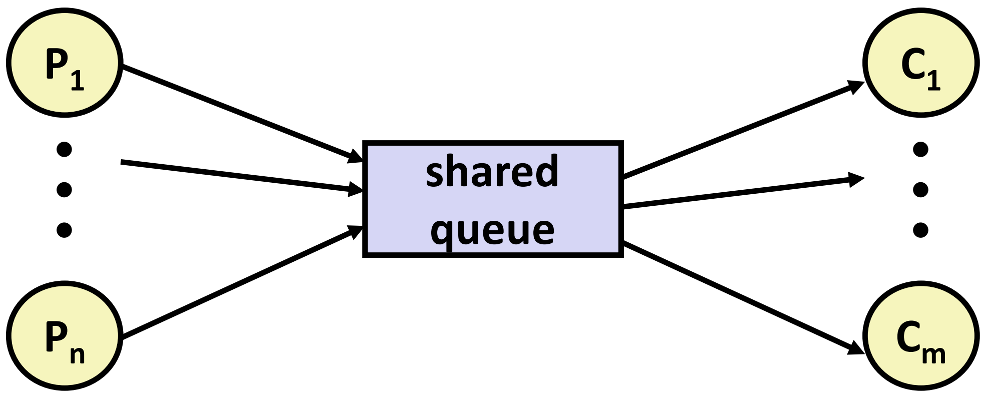
Producer-Consumer on 1-entry Queue

- Maintain two semaphores: `full` + `empty`



Why 2 Semaphores for 1-entry Queue?

- Consider multiple producers & multiple consumers



- Producers will contend with each to get `empty`
- Consumers will contend with each other to get `full`

Producers

```
P(&shared.empty);
shared.buf = item;
V(&shared.full);
```

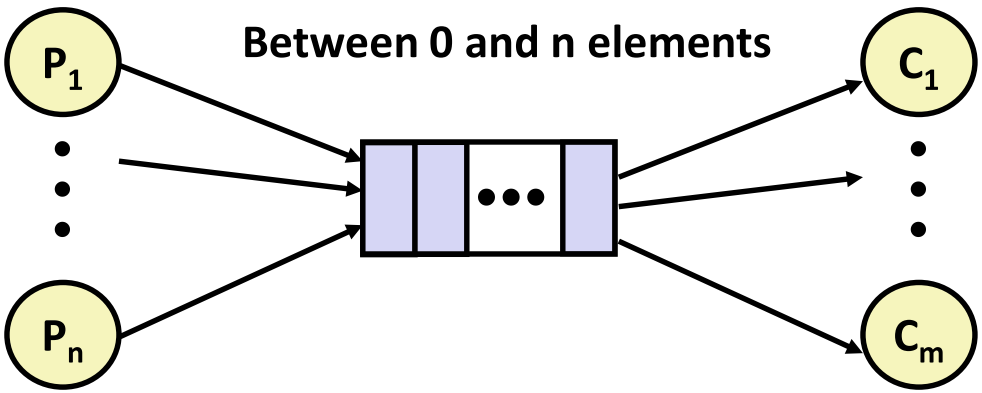
`empty`

`full`

Consumers

```
P(&shared.full);
item = shared.buf;
V(&shared.empty);
```

Producer-Consumer on *n*-element Queue

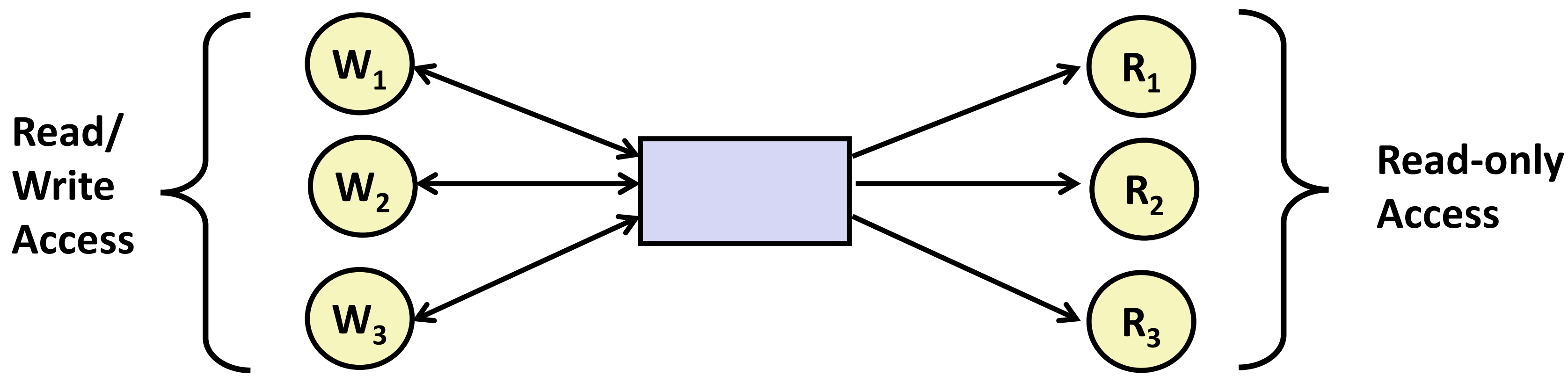


- **Requires a mutex and two counting semaphores:**
  - `mutex`: enforces mutually exclusive access to the queue's innards
  - `slots`: counts the available slots in the queue
  - `items`: counts the available items in the queue
- **Makes use of semaphore values > 1 (up to *n*)**

Today

- Deadlock
- Semaphores, Events, and Queues
- **Reader-Writer Locks and Starvation**
- Thread-Safe API Design

Readers-Writers Problem



- **Problem statement:**
  - *Reader* threads only read the object
  - *Writer* threads modify the object (read/write access)
  - Writers must have exclusive access to the object
  - Unlimited number of readers can access the object
- **Occurs frequently in real systems, e.g.,**
  - Online airline reservation system
  - Multithreaded caching Web proxy

Pthreads Reader/Writer Lock

- **Data type `pthread_rwlock_t`**
- **Operations**
  - Acquire read lock

```
pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
```

  - Acquire write lock

```
pthread_rwlock_wrlock(pthread_rwlock_t *rwlock)
```

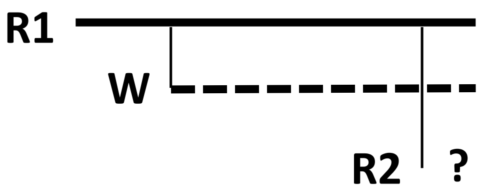
  - Release (either) lock

```
pthread_rwlock_unlock(pthread_rwlock_t *rwlock)
```

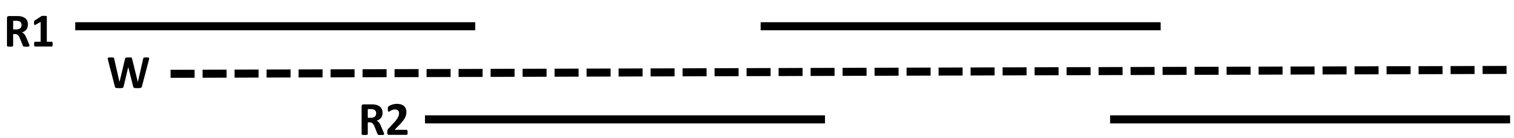
- **Must be used correctly!**
  - Up to programmer to decide what requires read access and what requires write access

Reader/Writer Starvation

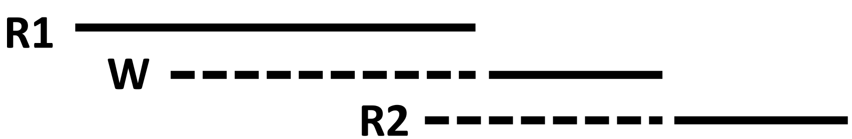
- **Thread 1 has a read lock. Thread 2 is waiting for a write lock. Thread 3 tries to take a read lock. What happens?**



- **Option 1: R2 gets read lock immediately**
  - Endless stream of overlapping readers → W waits forever



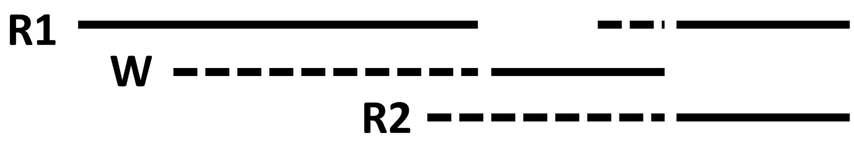
- **Option 2: Writer always gets lock as soon as possible**
  - Endless stream of overlapping writers → readers wait forever (not shown)



Starvation

- **A thread is *starved* when it makes no forward progress for an unacceptably long time**
  - Unlike deadlock, it's possible for it to get unstuck eventually
  - "Unacceptably long" depends on the application

- **Algorithms that guarantee no starvation are called *fair***
  - Fair R/W locks: every waiter receives the lock in first-come first-served order (several readers can receive the lock at the same time)



  - Fairness is more complicated to implement
  - Fairness can mean *all* threads are slower than they would be in an unfair system (e.g. "lock convoy problem")

Today

- Deadlock
- Semaphores, Events, and Queues
- Reader-Writer Locks and Starvation
- **Thread-Safe API Design**

Thread-Safe APIs

- **A function is *thread-safe* if it always produces correct results when called repeatedly from multiple concurrent threads.**

- **Reasons for a function *not* to be thread-safe:**
  1. Internal shared state, no locking (e.g. your **`malloc`**)
  2. Internal state modified across multiple uses (e.g. **`rand`**)
  3. Returns a pointer to a static variable (e.g. **`strtok`**)
  4. Calls a function that does any of the above

Thread-Unsafe Functions (Class 1)

- **These functions *would* be thread-safe if they began with `pthread_mutex_lock(&l)` and ended with `pthread_mutex_unlock(&l)` for some lock L**

- **Good example: `malloc`, `realloc`, `free`**
  - Your implementation will crash if called from multiple concurrent threads
  - The C library's implementation won't; it has internal locks

- **Locking slows things down, of course**

Thread-Unsafe Functions (Class 2)

- **Relying on persistent state across multiple function invocations**
  - Example: Random number generator that relies on static state

```
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

- **Difference from class 1: locking would not fix the problem**
  - 2 threads call rand() simultaneously, both get different results than if only one made a sequence of calls to rand()

Fixing Class 2 Thread-Unsafe Functions

- **Pass state as part of argument**
  - and, thereby, eliminate static state

```
/* rand_r - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- **Requires API change**
- **Callers responsible for allocating space for state**

Thread-Unsafe Functions (Class 3)

- **Returning a pointer to a static variable**
- **Like class 2, locking inside function would not help**
  - Race between *use of result* and calls from another thread
- **Fix: make caller supply space for result**
  - Requires API change (also like class 2)
  - Can be awkward for caller: how much space is required?

```
/* Convert integer to string */
char *itoa(int x)
{
    static char buf[11];
    snprintf(buf, 11, "%d", x);
    return buf;
}
```

```
/* Convert integer to string
   (thread-safe) */
void itoa_r(int x, char *buf,
            size_t bufsz)
{
    snprintf(buf, bufsz, "%d", x);
}
```

Thread-Unsafe Functions (Class 4)

- **Calling thread-unsafe functions**
  - Any function that uses a class 1, 2, or 3 function internally is just as thread-unsafe as that function itself
  - This applies transitively

- **Only fix is to modify the function to use only thread-safe functions**
  - This may or may not involve API changes

Thread-Safe Library Functions

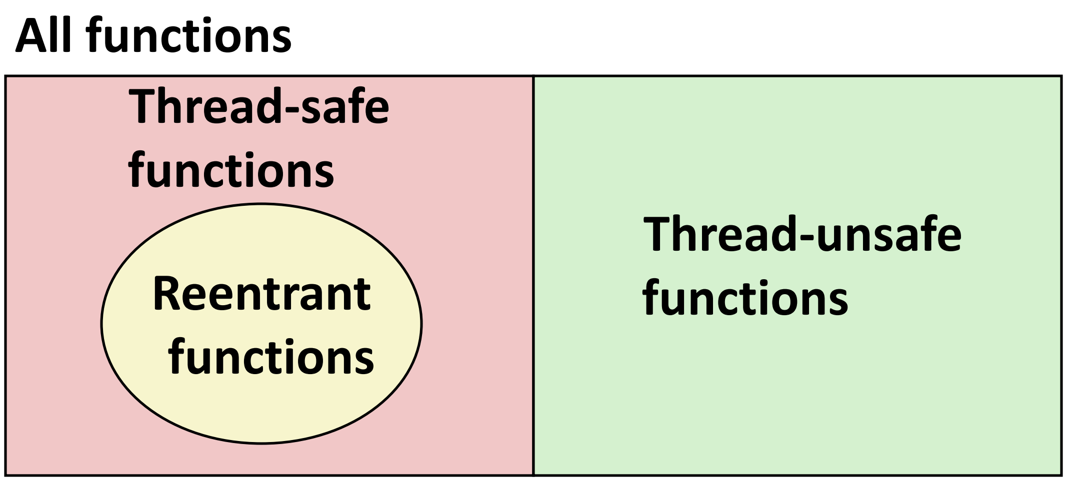
- **Most ISO C library functions are thread-safe**
  - Examples: `malloc`, `free`, `printf`, `scanf`
  - Exceptions: `strtok`, `rand`, `asctime`, …
- **Many older Unix C library functions are unsafe**
  - There is usually a safe replacement

Thread-unsafe function	Class	Reentrant version
`asctime`	3	`strftime`
`ctime`	3	`strftime`
`localtime`	3	`strftime`
`gethostbyname`	3	`getaddrinfo`
`gethostbyaddr`	3	`getnameinfo`
`inet_ntoa`	3	`getnameinfo`
`rand`	2	`rand_r*`

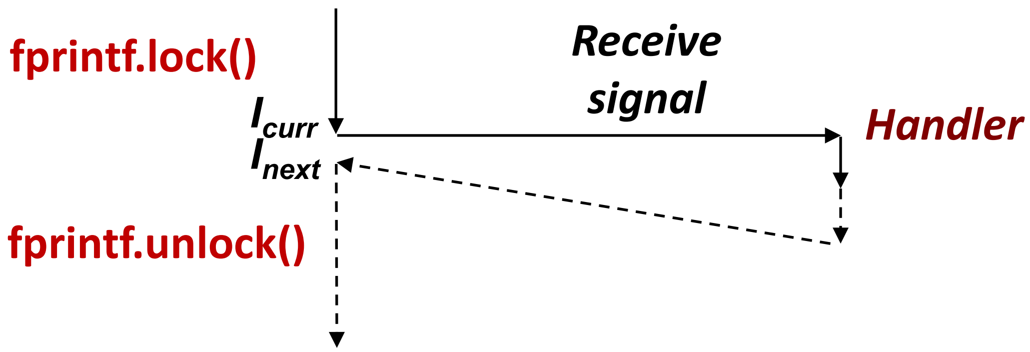
** The C library's random number generators are all old and not very "strong". Use a modern CSPRNG instead.*

Reentrant Functions

- **Def: A function is *reentrant* if it accesses no shared variables when called by multiple threads.**
  - Important subset of thread-safe functions
  - Require no synchronization operations
  - Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

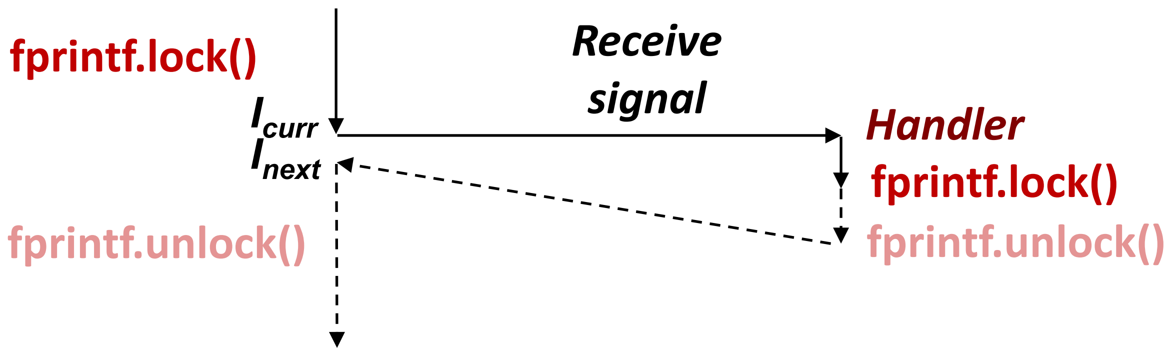


Threads / Signals Interactions



- **Many library functions use lock-and-copy for thread safety**
  - malloc
    - Free lists
  - fprintf, printf, puts
    - So that outputs from multiple threads don't interleave
  - snprintf
    - Calls malloc internally for scratch space
- **OK to interrupt them with locks held**
  - … as long as the handler doesn't use them itself!

Bad Thread / Signal Interactions



- **What if:**
  - Signal received while library function holds lock
  - Handler calls same (or related) library function
- **Deadlock!**
  - Signal handler cannot proceed until it gets lock
  - Main program cannot proceed until handler completes
- **Key Point**
  - Threads employ symmetric concurrency
  - Signal handling is asymmetric